

# Introducing BDDs in the Explicit-State Verification of Java Code

**jpf-bdd**

Alexander von Rhein, Sven Apel

Franco Raimondi

University of Passau  
Germany

Middlesex University, London  
UK



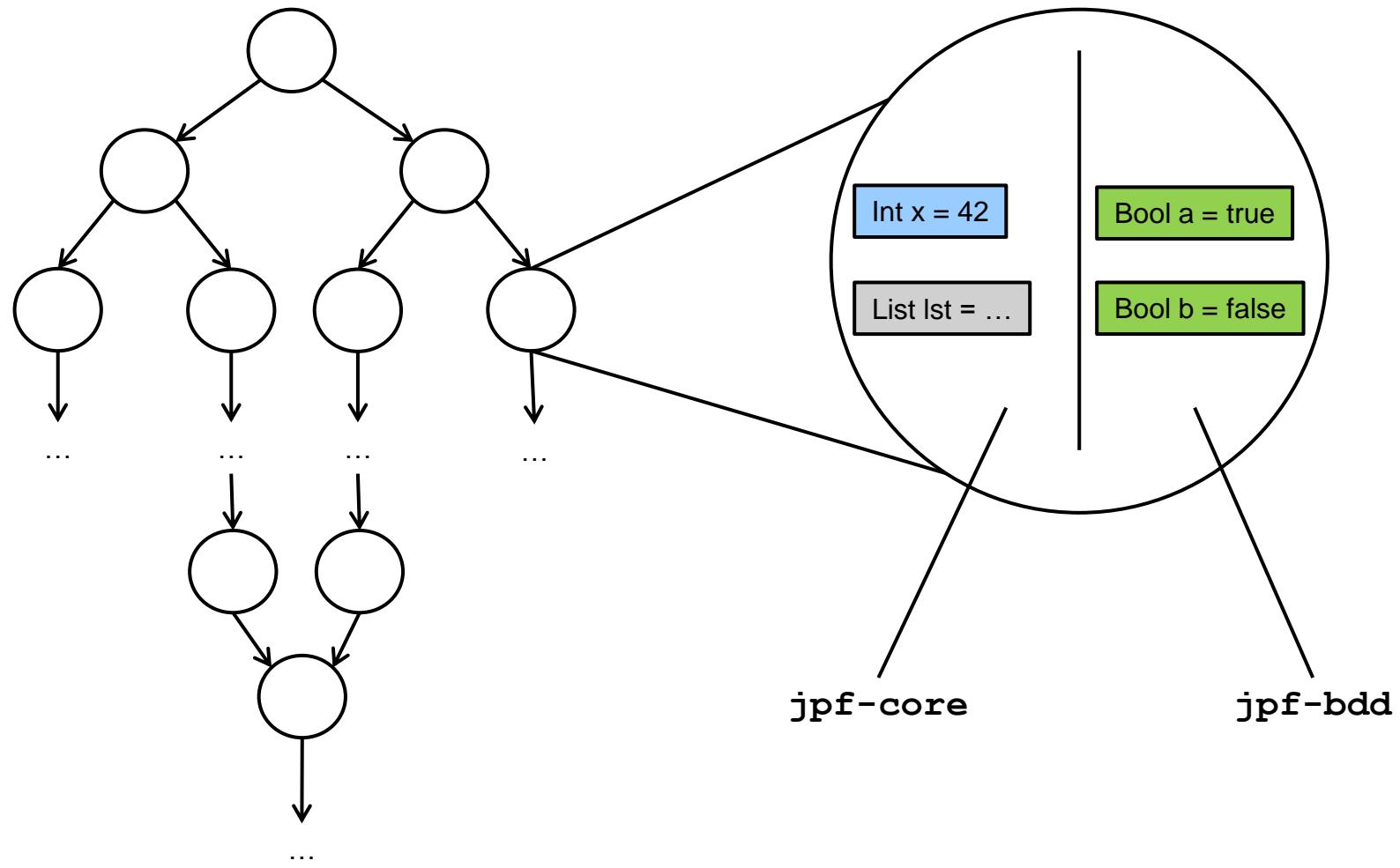
Software  
Product-Line  
Group



# Contents

- Motivation & Background
- **jpf-bdd**
- Experimental results

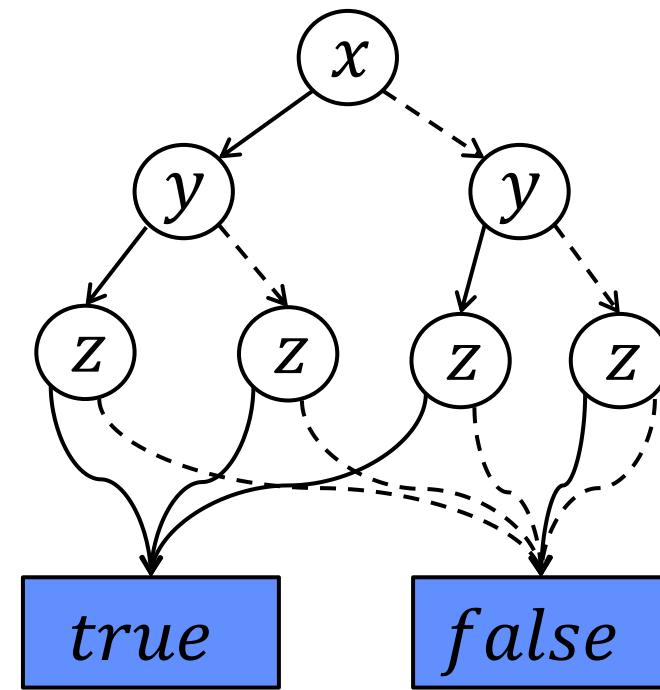
# Motivation



# Binary Decision Diagrams (BDD)

$$f(x, y, z) = (x \vee y) \wedge z$$

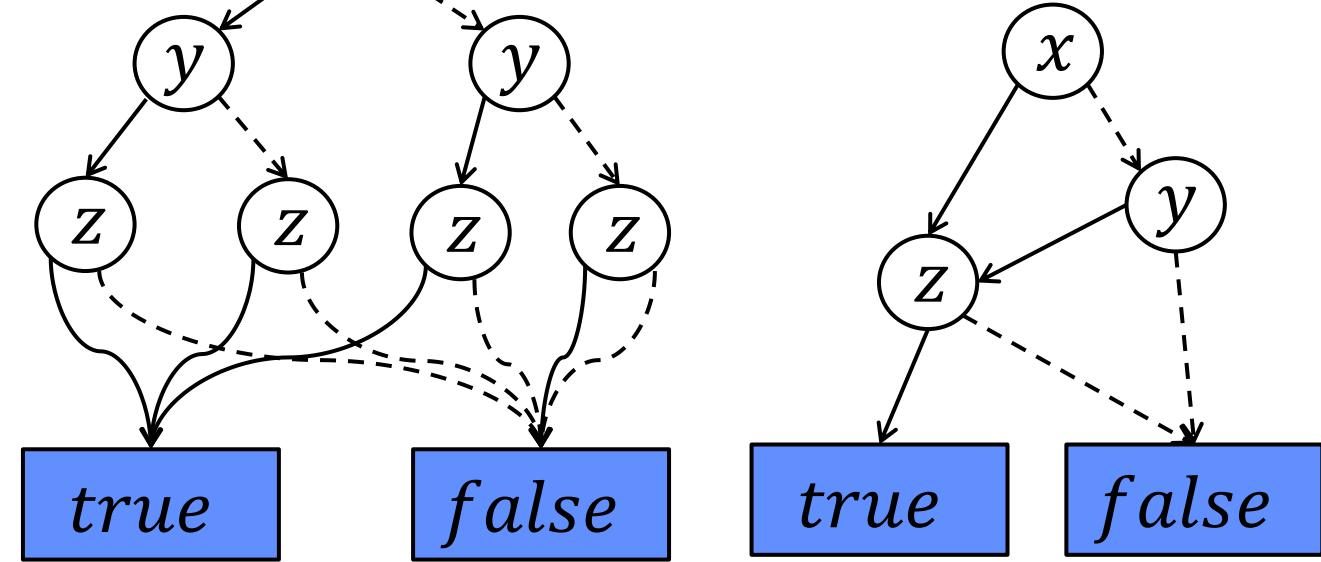
$x$	$y$	$z$	$f(x, y, z)$
1	1	1	1
1	1	0	0
1	0	1	1
1	0	0	0
0	1	1	1
0	1	0	0
0	0	1	0
0	0	0	0



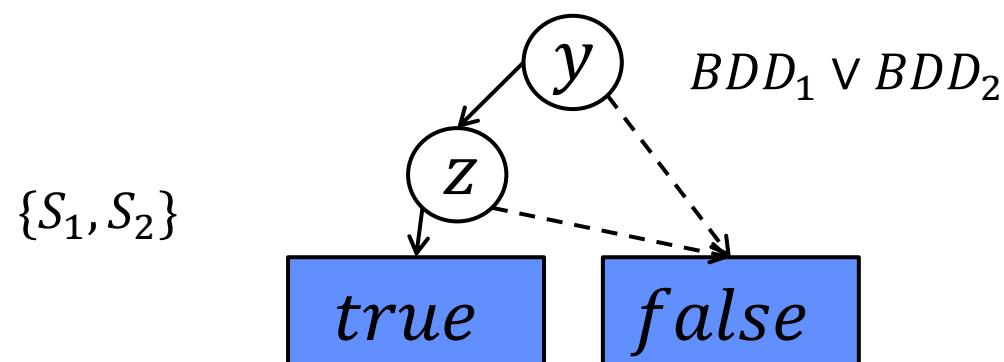
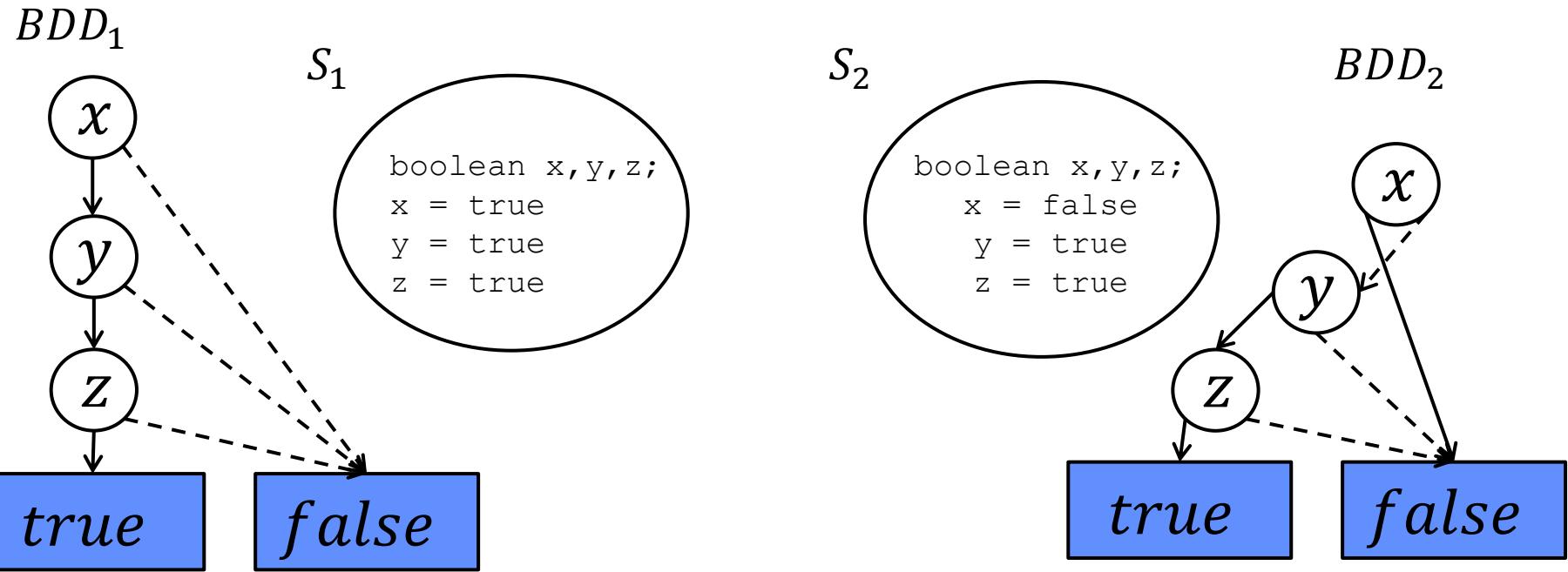
Truth Table

Ordered BDD

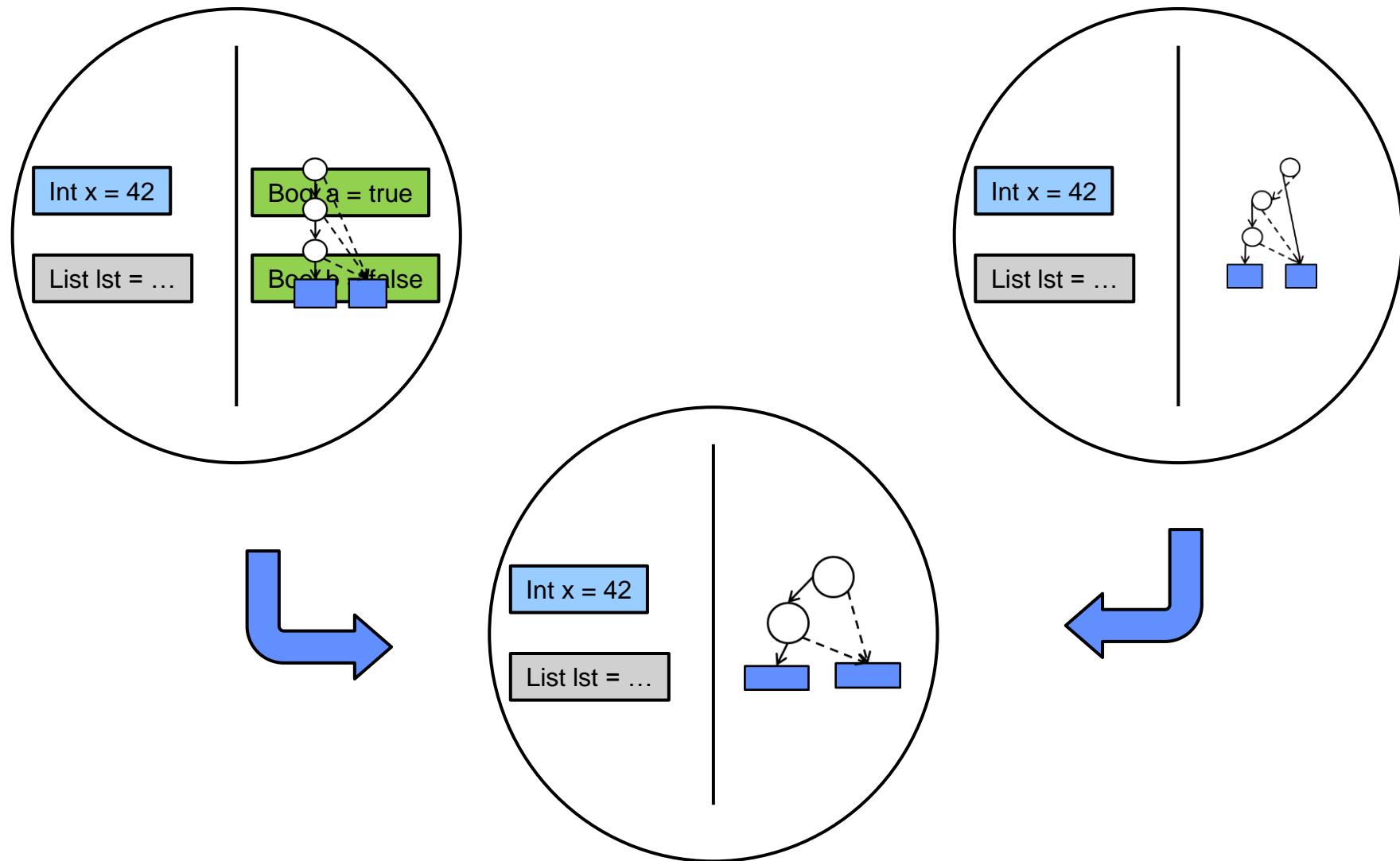
Reduced Ordered BDD



# Representing States with BDDs



# Our Approach



# Contents

- Motivation & Background
- **jpf-bdd**
- Experimental results

# The jpf-bdd extension

bar.java

```
@TrackWithBDD  
public boolean a;  
  
public void foo() {  
    if (a)  
        throw new Error();  
}
```

config.jpf

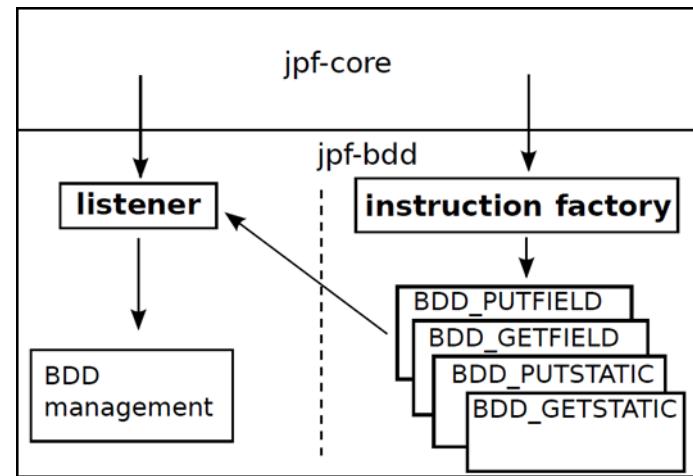
```
@using = jpf-bdd  
...  
↑
```

Search strategy is set to  
**Breadth-First-Search**

```
~$ JPF/jpf-core/bin/jpf config.jpf
```

# Implementation

- Listener
  - ◆ Holds the current BDD
  - ◆ Manages BDDs of search states
  - ◆ Handles backtracking
- Instruction factory
  - ◆ Special instructions for GETFIELD / GETSTATIC and PUTFIELD / PUTSTATIC
- Instructions
  - ◆ Modify the current BDD (in the listener)
  - ◆ Install ChoiceGenerators if necessary (next slide)

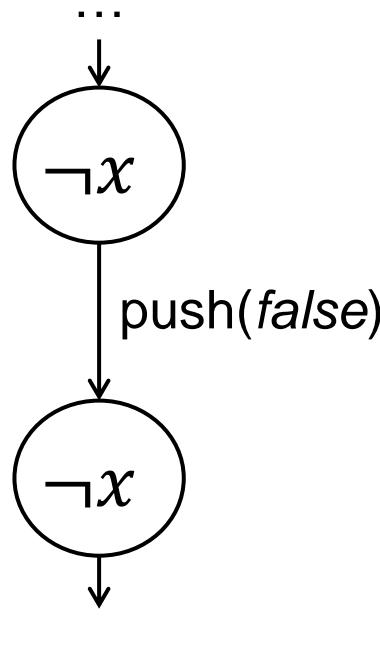


# Handling the byte-code instructions

- GETFIELD (other instructions are handled similarly)
  - ◆ Push the variable value on the stack

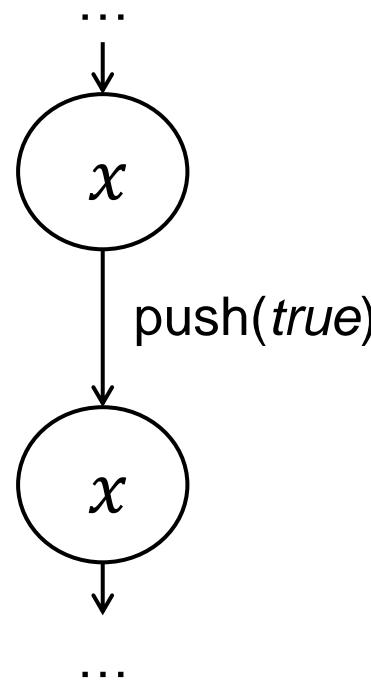
Only *false* is possible

$$\neg \text{SAT}(\text{BDD} \wedge (\neg x))$$

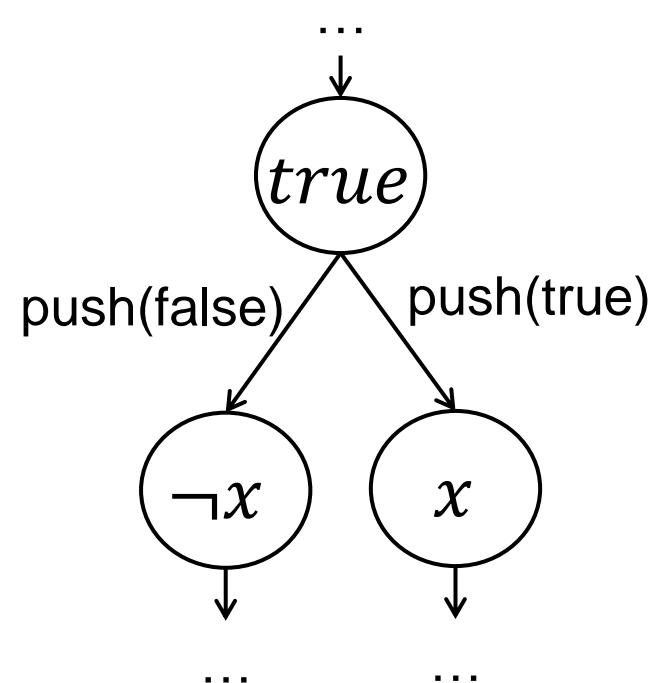


Only *true* is possible

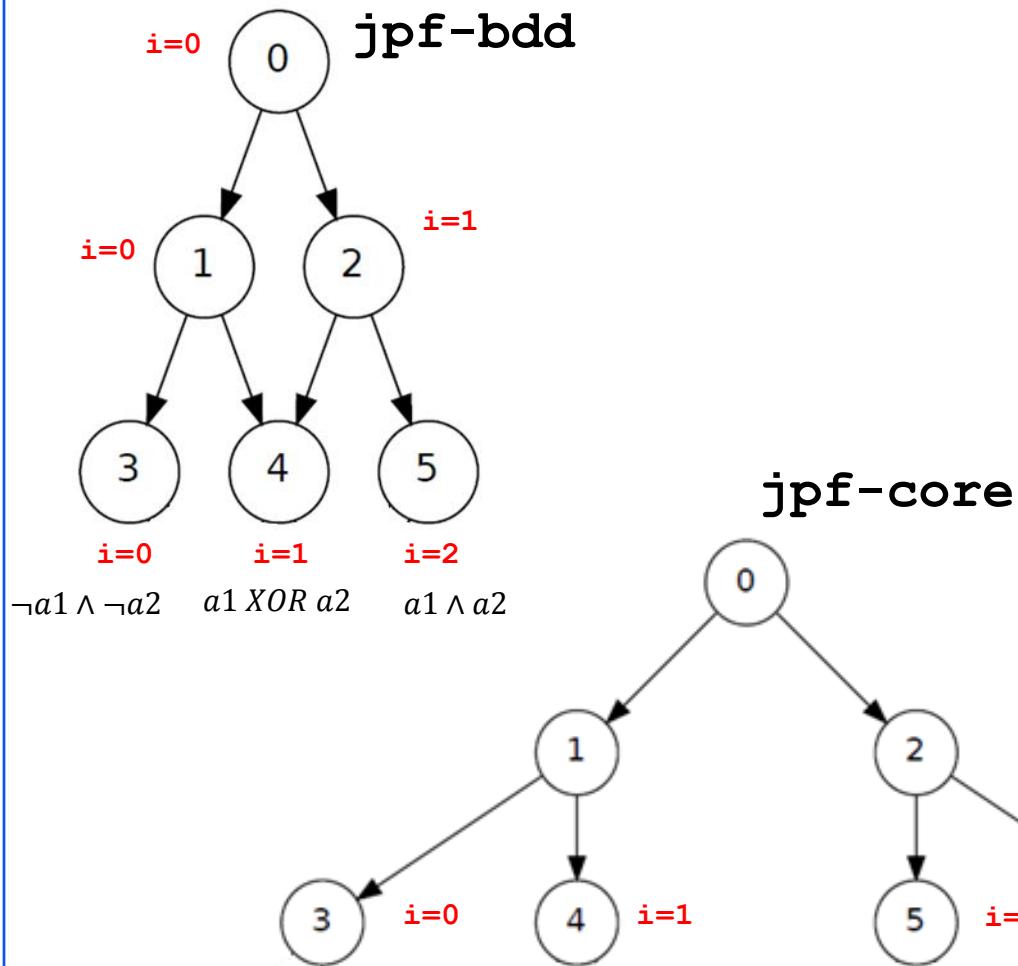
$$\neg \text{SAT}(\text{BDD} \wedge (x))$$



Both are possible  
*else*



# A simple program verification

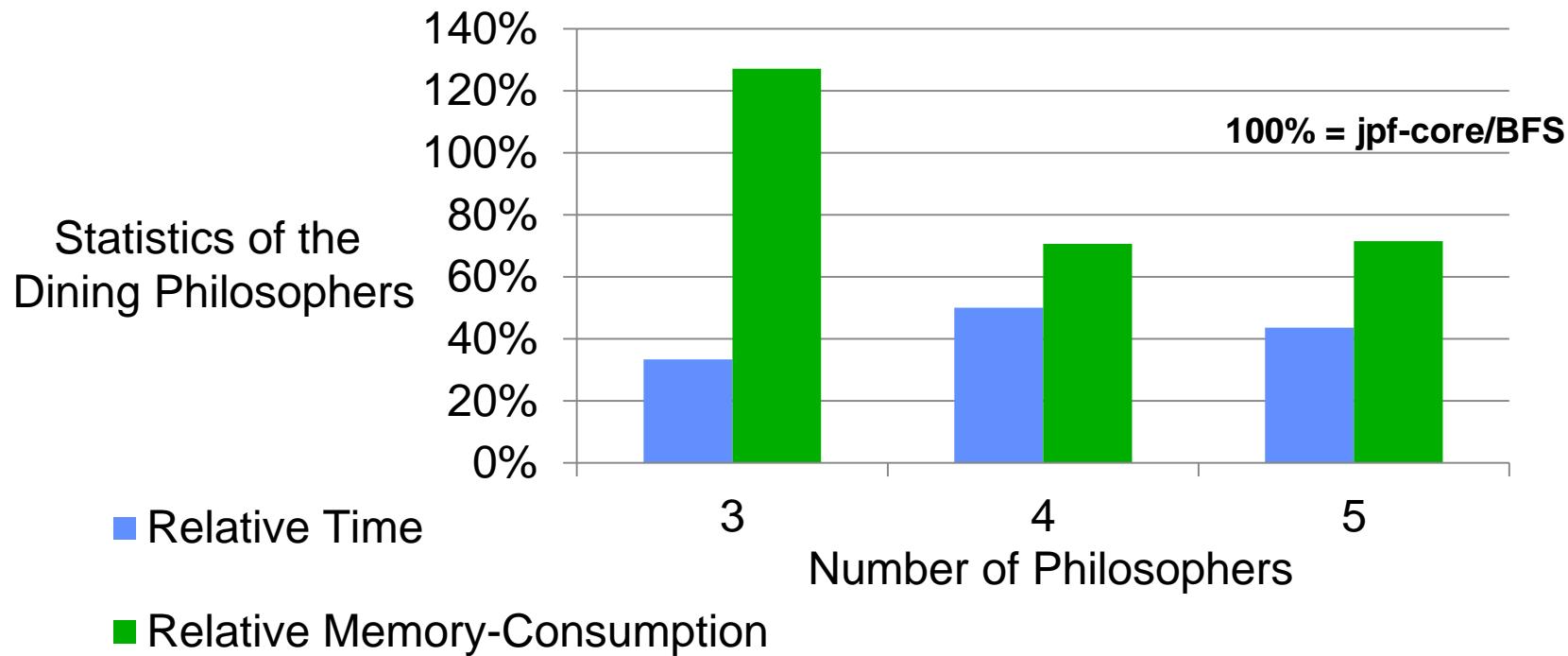


```

1 @TrackwithBDD
2 boolean a1, a2;
3 int i = 0;
4
5 public void foo() {
6   if (a1 = getBoolean()) {
7     i++;
8   }
9   if (a2 = getBoolean()) {
10    i++;
11  }
12  Verify.breakTransition();
13  System.out.println(i);
14 }
  
```

# Preliminary Experiments

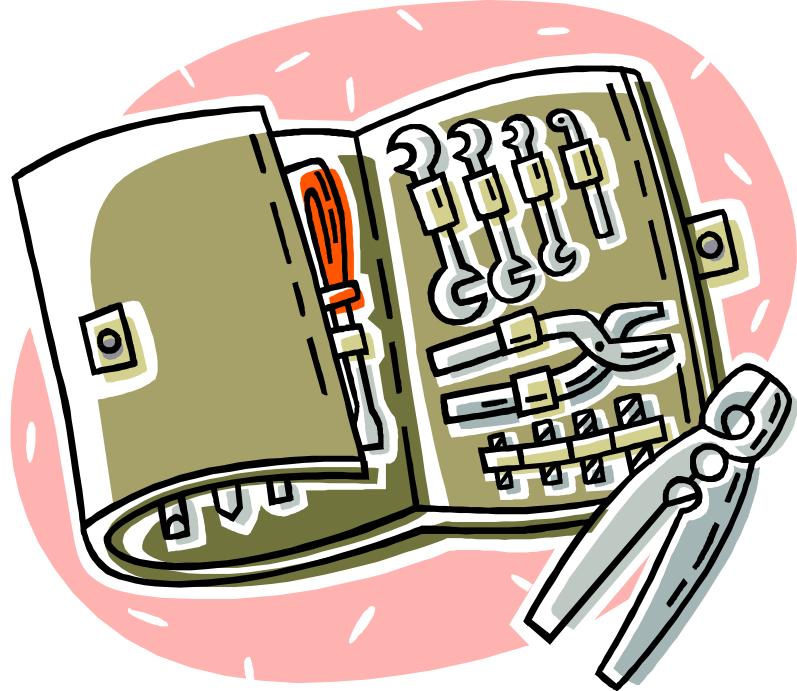
- Three sample programs
  - ◆ Two simple examples for detailed analysis
  - ◆ Implementation of the dining philosophers (multi-threaded)



# Observations

1. **jpf-bdd** needs more memory than **jpf-core/DFS**
  - ◆ Because we use BFS (**jpf-core/BFS** needs even more)
2. **jpf-bdd** creates fewer states (normally)
  - ◆ Because paths are merged
3. **jpf-bdd** executes fewer instructions
  - ◆ Because paths are merged and executed only once
4. **jpf-bdd** is faster (58% in the philosophers example)
  - ◆ The verification time has an almost linear relation to the number of executed instructions.

# Demo



# Conclusions

- Easy to use
- Faster when
  1. Many program states differ in the values of some boolean variables (e.g., configuration flags)
  2. The user can identify these variables
  3. You have enough memory to use BFS (and save time!)

# Questions

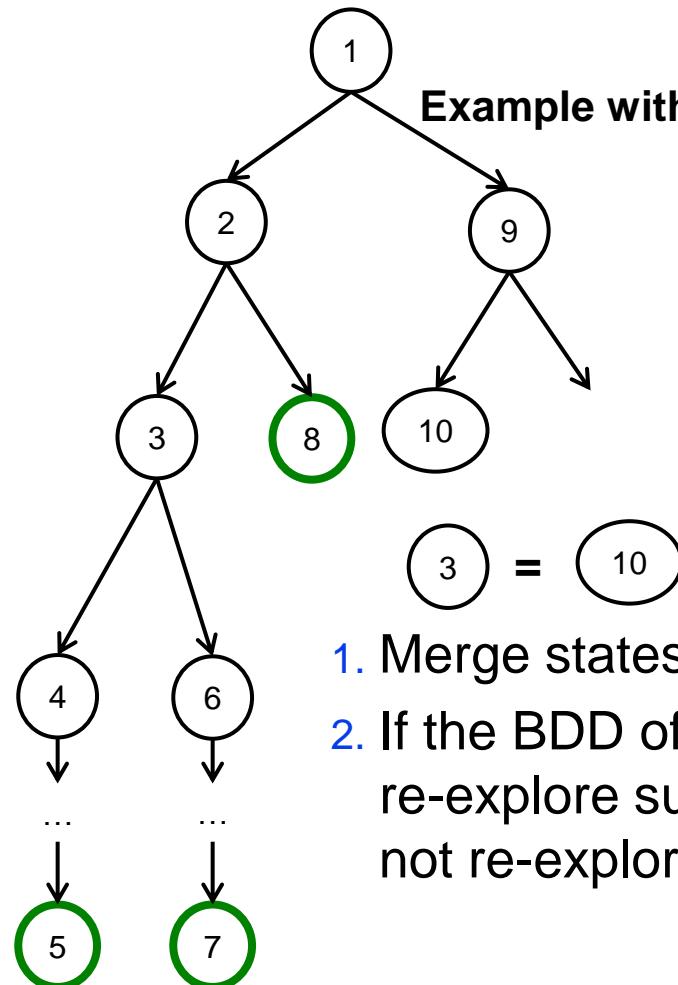


**jpf-bdd**

<http://bitbucket.org/rhein/jpf-bdd/>

JPF extension with  
Binary Decision Diagrams

# Why to use Breadth-First-Search?



1. Merge states 3 and 10
  2. If the BDD of {3,10} is different from the BDD of state 3: re-explore subtree of state 3 (if it is not different, then do not re-explore)
- BFS needs always much more memory than DFS